CMPE 365 Project – Recurrence Problem

This project was completed using C++11.

For future reference:

unordered\_map in C++11 is essentially a hash table.

M is the number of planes.

N is the number of unique people.

P is the average plane size.

Co-occurrences equal to the threshold value are included in the interesting co-occurrence list.

# Algorithm

For each plane, the algorithm iterates through each pair in the plane using nested for loops. A table is used to count the number of times each pair occurs, using each person’s ID as the location to increment. When the value at this table location is at the threshold, the pair is inserted into an unordered\_map. The person IDs are concatenated as a string to be the key and the count is the map value. This key is unique as the person IDs are all unique, and no person will co-occur with them self.

Optionally, when that pair key is already in the map, the value at that location is incremented so that the map contains the actual number of times the pair occurred, for completeness. However, the given question only asks for a list of the pairs that co-occur above some threshold; to save execution time, the true count is not included in the unordered\_map. This is discussed in more detail in Appendix A.

After each plane is computed, the unordered\_map is simply iterated through to output each pair that re-occurred the given number of times (or greater).

A key detail in this algorithm is that the planes of people are all sorted in order of person ID. That way, starting from the beginning of a plane and iterating to the end to find each pair will never result in inverse pairs. The people i and j, where i < j, will always be interpreted as (i,j) and not (j,i) as i is always found with the iterations before j. If this were to not hold, the implemented algorithm would need to be tweaked in order to work. This is discussed in Appendix B.

A small note is that a threshold of 0 breaks the program. This is discussed in detail in Appendix B.

# Algorithm Complexity

Clearly, any solution to this problem must examine each person on each plane at least once. That means the absolute optimal solution must have a complexity of O(MP) or worse. This is useful to compare with the created algorithm complexities. Another piece of interesting information is that since there are N people, P must be less than or equal to N.

## Implemented Algorithm Complexity

The initial setup of the algorithm is to create an unordered\_map and a zeroed 2D N by N matrix. These complexities are both O(1); the complexity of the matrix initialization is discussed in Appendix A.

Then the algorithm iterates through each plane in the data set, which is O(M). For each plane, every passenger pair is found using a nested for loop in the form:

*for i = 0 to plane size*

*for j = i+1 to plane size*

This is O(P2). For each pair found, their co-occurrence count is incremented. This is O(1) because the co-occurrence table uses the unique person IDs as the indices of the pair’s co-occurrence count. Then, that count is checked against the threshold value. Should it equal (and only equal, in this implementation as discussed above) the threshold, a key is made using the pair’s person IDs and inserted into an unordered\_map. This is all O(1) as well. The complexity of accessing the map to insert/update is discussed in Appendix A. The key for the map is a string type, made with the to\_string() function; the to\_string() complexity is also discussed in Appendix A. For this analysis, the unordered\_map access and the to\_string() function are being treated as O(1).

After those iterations, all pairs that co-occur more than or equal to the threshold will be in an unordered\_map. This completes the task and thus the algorithm. Therefore, the complexity of the algorithm is O(1) + O(M) \* O(P2) \* [O(1) + O(1)], as for each pair the O(1) operations are completed and the each pair is found for each plane. Therefore, the overall complexity is (MP2).

## P Compared to N

The implemented algorithm complexity is O(MP2). Technically, P is a function of N, as P <= N, so P will equal N/K, where K is some constant. For example, a decent guess would be P = N/2. Therefore, O(P) is really O(N/K), which falls to O(N). Thus, one can argue the Big-O of the algorithm is actually O(MN2). This is true for the worst possible case of P = N, although we know in practice that is not likely true.

# Speed

To increase the speed of the algorithm’s execution, no information was printed. Console output takes a relatively long time and is not relevant to the algorithm’s efficient (the problem does not ask for a printed output). Another note is that, again, the true count of a pair’s co-occurrence is not found once they reach the threshold, as discussed above. Finally, the time to read in the csv file is not included, as the algorithm to solve the problem does not involve how the data is obtained.

## Design Choices

Several design choices were made in order to improve the execution time. First, pre-increment is always used instead of post-increment. This is because post-increment requires copying the variable, making more computations than pre-increment. The second choice was to hold the plane size in a variable instead of using it directly in the nested for loops. This is so that the size() function is only called once instead of for each nested loop iteration. Finally, when an index is not needed in a for loop, a range for loop as used with a constant reference. In C++ this avoids copying a variable and therefore saving execution time. Unfortunately, this can only be done in the outer-most loop, as the inner loops need indices to systematically find each plane’s pairs.

Other design choices include avoiding finding the true co-occurrence count after the threshold is reached, and using a static pair count matrix to avoid initialization costs. These are both discussed in Appendix A.

## Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **lists.csv** | | | **newlists.csv** | | |
| **Threshold** | **Average of 10 runs (ms)** | **Interesting Pairs** | **Threshold** | **Average of 10 runs (ms)** | **Interesting Pairs** |
| 1 | 207.3 | 44850 | 1 | 218.5 | 44850 |
| 5 | 206 | 44849 | 5 | 214 | 44847 |
| 10 | 200.3 | 44471 | 10 | 220.1 | 44052 |
| 25 | 17.47 | 1269 | 25 | 75.3 | 14983 |
| 45 | 9.7 | 0 | 45 | 12.3 | 1318 |
| 100 | 2.2 | 0 | 100 | 5.3 | 0 |

Clearly, the speed of execution increases as the threshold increases. As the threshold grows, less operations are conducted with the unordered\_map.

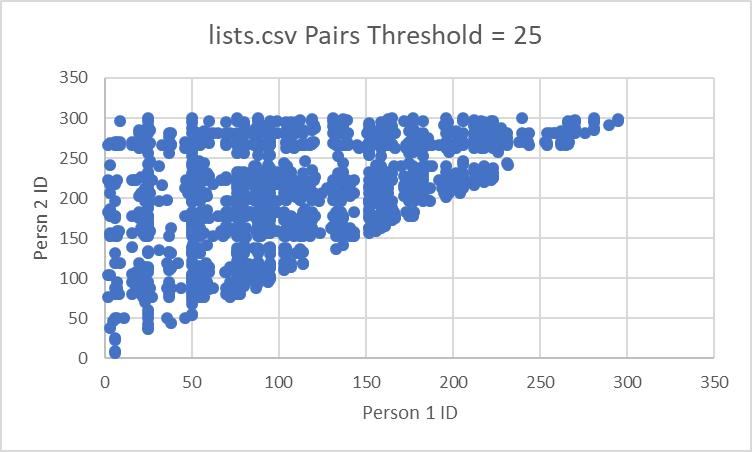
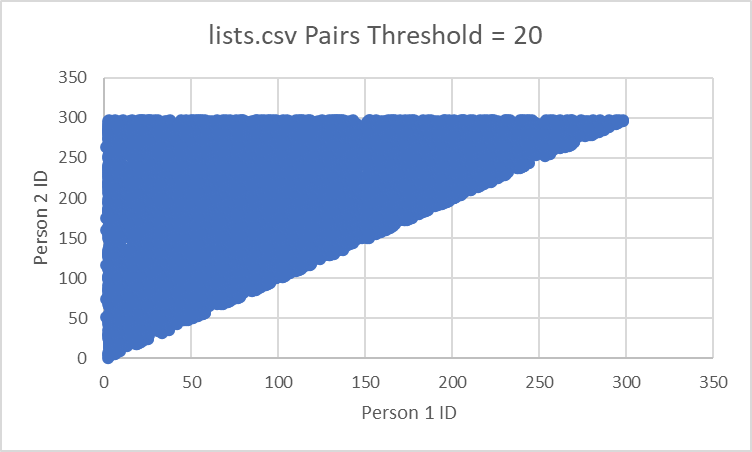
The execution speed of newlists.csv is consistently higher than the time for lists.csv. Looking at the two files, it appears that the average plane size is larger in newlists.csv, which would increase execution time as the algorithm is O(MP2).This is backed up by the higher number of interesting pairs for each threshold (except 1) in newlists.csv.

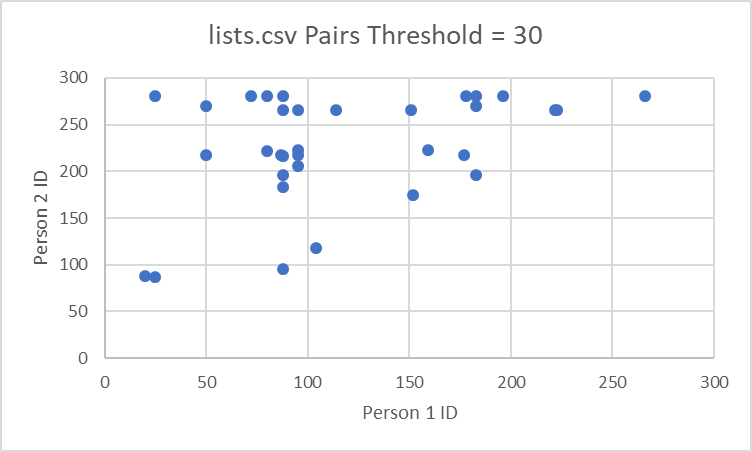
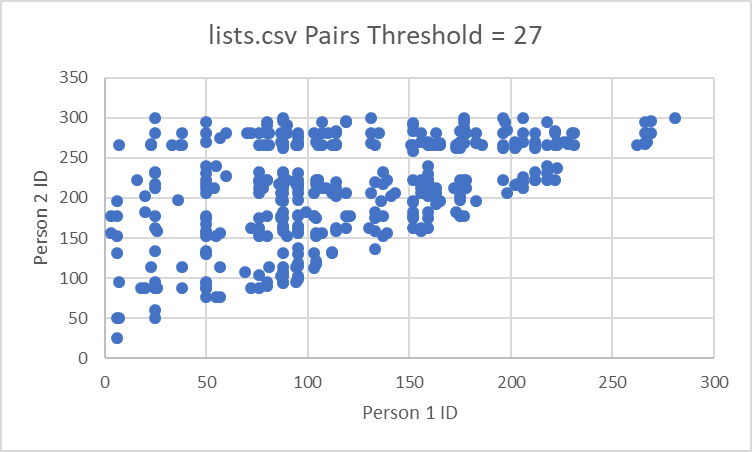
# Visualizing Change with Threshold

Even though I am working alone, I chose to attempt the extra part of the project where the change in data should be visualized as the threshold changes.

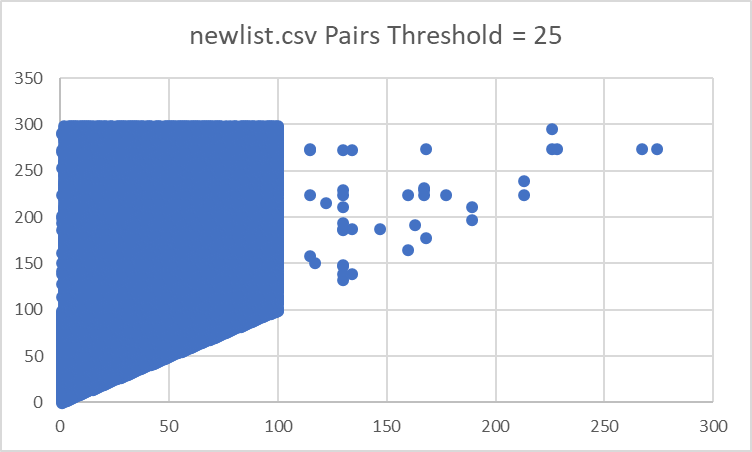
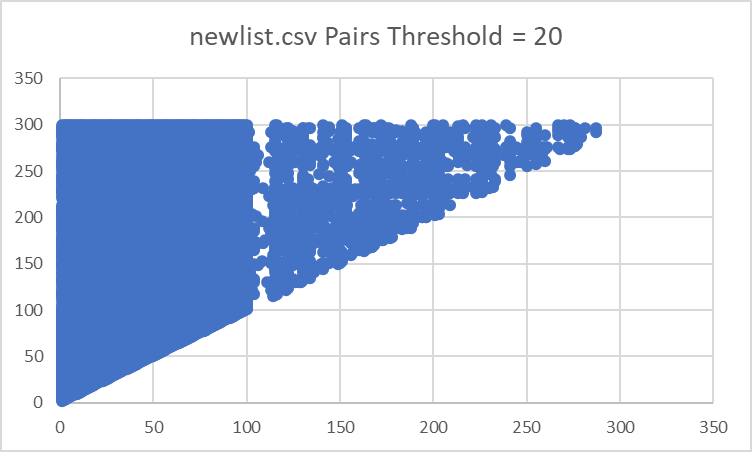
To visualize, the ID of the first person in the pair is plotted against the ID of the second person. As the threshold increases, the density of points will decrease. Also, there will never be a point beneath the y=x line, as that would be an inverse pair.

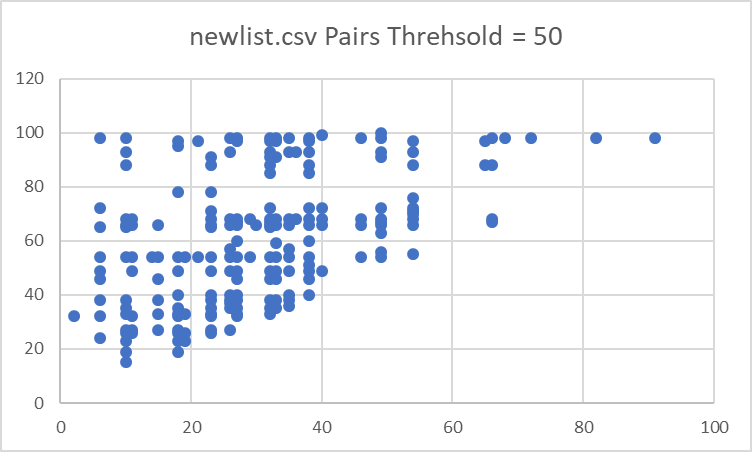
### Lists.csv





### Newlists.csv





# Appendix A

## Tracking Number of Co-occurrences

In the current code, the actual number of co-occurrences are not tracked in the unordered\_map. Once a pair reaches the threshold value they are inserted into the map and the count is not updated further. This is simply to avoid extra accesses to the table holding the true count of each pair’s co-occurrences. The question posed only requires a list of the pairs above the threshold and not the actual number of co-occurrences.

If the true co-occurrence count is required, only a simple change to the code is needed, as outlined in the comments where a pair is inserted into the unordered\_map.

If only the pairs are needed, not the true count, then a different data structure such as a vector would more specifically suited to the task; however, there is no visible difference in computation speed or number of commands when an unordered\_map is used. Therefore, the map is implemented simply to increase the ease of switching to taking the true pair count.

## Complexity of unordered\_map Access

The unordered\_map is inserted into and accessed using the [] operator. When this operator is used with the key of an entry supplied as a parameter, the corresponding value is returned. If the supplied key is not in the map, that key is inserted with some default value (the default is based on the value type). The complexity of this operator is constant in the average case and linear based on the map size in the worst case. This worst case is only reached when an element being inserted/accessed has the same key as other elements in the unordered\_map. However, because the key created in the implementation is simply “<first person’s ID>,<second person’s ID>”, and the IDs are all unique, the key will always be unique, which should lead to a unique store in the map. So for this usage, the unordered\_map access and insertion complexities will always be O(1).

Having said that, inserting new elements may cause a rehash in the map. This would increase the complexity. This is assumed to not happen often and thus for this analysis, ignored.

## Complexity of to\_string()

After conducting quite a bit of research, the complexity of the to\_string(int) function in C++ could not be found. The function was speed benchmarked against 9-digit numbers and found to run in 0 milliseconds, even when adding a comma to the result (as the implemented algorithm does). As a result, a safe assumption is that this function does not greatly impact the runtime of the implemented algorithm. For this analysis, the complexity is assumed to be O(1).

This is also only used to create a unique key for the co-occurred pair, which could be done in a different way if this function were too slow.

## Complexity of Matrix Initialization

In C++11, a 2D array can be initialized with values at every position using the {} initializer. In theory, this appears to be O(N2); however, compilers should optimize this to be quite efficient.

In the worst possible case, the initialization of a matrix to 0 will be a nested for loop used to set each position to hold 0. This would be an O(N2) operation as the matrix used is N by N. This would raise the implemented algorithm complexity to O(N2 + MP2), which, unless M and/or P are much smaller than N, would result in the same O(MP2) overall. The time of execution of this operation was tested using a 500 by 500 matrix (larger than the 300 by 300 matrix used in this problem) was 0 milliseconds, so clearly this has no impact on this specific problem dataset.

A workaround is to declare the 2D array as static. This makes every value automatically be initialized to 0, saving quite a bit of computation time. This does create a memory leak as the table should not be created for the entire program life; however, in this case the table is only made once, reducing the memory leak problems.

Using a static table, or assuming the compiler is very effective in optimizing initialization, the creation of the zeroed table is O(1).

# Appendix B

## Threshold of 0

A small note is that a threshold of 0 breaks the program. To avoid unneeded accesses, the condition to insert after incrementing a pair count is that that count is equal to the threshold. This is never true with a threshold of 0. The fix is to set this to be greater than or equal to 0, although it will add a lot of unneeded computations. A better fix would be to make a special case: if the threshold is 0, every pair is interesting.

## Unsorted Planes

Several changes could be made to handle planes that are not sorted in order of person ID.

### Tweaking the Current Algorithm

The immediate possibility is to simply sort each plane by person ID instead of relying on the input being sorted. This however comes at the cost of O(PlogP) complexity. That would raise the overall complexity to O(MP3logP), which is unacceptable.

The much better option would be to allow the inverse pairs to be counted. Then, when the current pair threshold check is performed, the inverse pair is added to the current pair, and the total is compared to the threshold. This would add computations to access the second pair, but it is likely the best option in terms of ease of change and computation.

## Using the Marking Algorithm

This algorithm uses the memoization concept from dynamic programming with respect to who is present within a given plane. Rather than iterating through each plane in nested for loops, a plane is iterated through once, marking which people are present in a table. This table is simply a matrix, initialized at 0s, of each possible person ID – 1 (ID’s are unique, -1 as arrays in C++ are indexed at 0). Marking is simply incrementing the value at markingTablle[i][personID-1] and at markingTable[personID-1][i]. When i is equal to personID-1, no marking is done as this would be the same person in the pair. Should an entry in the table have 2 after a new marking, that means there is a pair present in the plane. An unordered\_map is used to store that pair as a key (string concatenated, unique as the person IDs are unique), with the value as 1 (the number of times they have been on a plane together. If the map already holds that key, instead of insertion, the value at that key is incremented.

This is done for each plane with a newly zeroed marking matrix. Therefore, after all planes have been computed, the unordered\_map holds all recurring pairs and the number of times they re-occur. The map is then iterated through to find the pairs, based on the key value, that re-occur the given number of times (or greater).

To avoid extra compute time of searching for markings of 2 after marking is over, each time a marking is made, that newly updated location is checked for a marking of 2. If it has a marking of 2 it is immediately inserted into the unordered\_map.

The complexity of this marking algorithm is O(MP)+O(L), where L is the number of pairs, as each plane is iterated through once, and then at the end the unordered\_map is traversed to find pairs above the threshold. Based on the complexity, clearly the original algorithm should be tweaked and used instead of this one, but I thought this was an interesting approach.